Strategy-Enhanced Interactive Proving and Arithmetic Simplification for PVS

Ben L. Di Vito

NASA Langley Research Center, Hampton VA 23681, USA
b.l.divito@larc.nasa.gov
http://shemesh.larc.nasa.gov/~bld

Abstract. We describe an approach to strategy-based proving for improved interactive deduction in specialized domains. An experimental package of strategies (tactics) and support functions called Manip has been developed for PVS to reduce the tedium of arithmetic manipulation. Included are strategies aimed at algebraic simplification of real-valued expressions. A general deduction architecture is described in which domain-specific strategies, such as those for algebraic manipulation, are supported by more generic features, such as term-access techniques applicable in arbitrary settings. An extended expression language provides access to subterms within a sequent.

1 Introduction

Recent verification research at NASA Langley has emphasized extensive theorem proving over the domain of reals [4, 5], with PVS [15] serving as the primary proof tool. Efforts in this area have met with some difficulties, prompting a search for improved techniques for interactive proving. Significant productivity gains will be needed to fully realize our formal methods goals.

For arithmetic reasoning, PVS relies on decision procedures augmented by automatic rewriting. When a conjecture fails to yield to these tools, which often happens with nonlinear arithmetic, considerable interactive work may be required to complete the proof. Large productivity variances are the result.

SRI continues to increase the degree of automation in PVS. In particular, decision procedures for real arithmetic are a planned future enhancement. We look forward to these improvements. Nevertheless, there will always be a point where the automation runs out. When that point is reached, tactic-based[1] techniques can be applied to good effect.

In this paper we describe an approach to strategy-based proving for improved interactive deduction in specialized domains. An experimental package of strategies (tactics) and support functions called Manip has been developed for PVS to reduce the tedium of arithmetic manipulation. Included are strategies aimed at algebraic simplification of real-valued expressions. A general deduction architecture is described in which domain-specific strategies, such as those for algebraic manipulation, are supported by more generic features, such as term-access techniques applicable in arbitrary settings. User-defined proof strategies can be seen as a type of "deductive middleware." Our approach is general enough to serve other problem domains in the pursuit of such middleware.

By way of motivation, consider the following lemma for reasoning about trigonometric approximations:

$$0 < a \leq \pi/2 \supset |T_n(a)| > 2\,|T_{n+1}(a)| \tag{1}$$

where $T_i(a)$ is the ith term in the power series expansion of the sine function:

$$\sin(a) = \sum_{i=1}^{\infty} (-1)^{i-1} a^{2i-1}/(2i-1)! \ .$$

[1] In PVS nomenclature, a *rule* is an atomic prover command while a *strategy* expands into one or more atomic steps. A *defined rule* is defined as a strategy but invoked as an atomic step. For our purposes, we regard the terms "tactic," "strategy" and "defined rule" as roughly synonymous.

Using only built-in rules, an early proof attempt for (1) required 68 steps. A common technique to carry out algebraic manipulation in such proofs is to use the `case` rule to force a case split on the (usually obvious) equality of two subexpressions, such as:

```
(CASE "a!1 * a!1 * (b!1 * b!1) = (b!1 * a!1) * (b!1 * a!1)")
```
(2)

Although not peculiar to PVS, this need to identify equivalent subexpressions and bring them to the prover's attention via cut-and-paste methods is rather awkward. It leads to a tedious style of proof that tries the patience of most users.

In contrast, by using the Manip package we were able to prove the lemma more naturally in 18 steps, 8 of which are strategies from our package, as shown in Fig. 1. Unlike the case-split technique, none of the steps contains excerpts from the sequent, such as those seen in (2). This proof represents one of the better examples of improvement from the use of our strategies. Although many proofs will experience a less dramatic reduction in complexity, the results have been encouraging thus far.

```
("" (SKOSIMP*)
    (REWRITE "sin_term_next")
    (RECIP-MULT! (! 1 R (-> "abs") 1))      ; strategy
    (APPLY (REPEAT (REWRITE "abs_mult")))
    (PERMUTE-MULT 1 R 3 R)                  ; strategy
    (OP-IDENT 1 L 1*)                       ; strategy
    (CANCEL 1)                              ; strategy
    (("1" (EXPAND "abs")
          (ASSERT)
          (PERMUTE-MULT 1 R 2 R)            ; strategy
          (CROSS-MULT 1)                    ; strategy
          (MULT-INEQ -2 -2)                 ; strategy
          (TYPEPRED "PI")
          (EXPAND "PI_ub")
          (MULT-INEQ -4 -4)                 ; strategy
          (ASSERT))
     ("2" (USE "sin_term_nonzero")
          (GRIND NIL :REWRITES ("abs")))))
```

Fig. 1. Proof steps for lemma (1) using built-in rules plus manipulation strategies

2 Architecture

We have integrated several elements to arrive at a strategy-based deduction architecture for user enhancements to PVS.

1. *Domain-specific proof strategies.* Common reasoning domains, such as nonlinear real arithmetic, provide natural targets for increasing automation. Extracting terms from sequents using suitable access facilities is vital for implementing strategies that do meaningful work.
2. *Extended expression language.* Inputs to existing prover rules are primarily formula numbers and expressions in the PVS language. For greater effectiveness, we provide users with a language for specifying subexpressions by location reference and pattern matching.
3. *Higher-order strategies with substitution.* Strategies that apply other proof rules offer the usual convenience of functional programming. Adding command-line substitutions derived from sequent expressions yields a more powerful way to construct and apply rules dynamically.
4. *Prelude extension libraries.* The PVS *prelude* holds built-in core theories. Strategies use prelude lemmas but often need additional facts. PVS's prelude extension feature adds such theorems in a manner transparent to the user.

5. *User-interface utilities.* To improve command line invocation of proof rules as well as offer various proof maintenance functions, a set of Emacs-based interface enhancements is included.

Note that only elements 1 and 4 are domain specific; the others are quite generic. In this paper we will focus on elements 1–3.

Several benefits accrue from the complementary elements of this architecture.

- User interaction is more natural, less laborious and occurs at a higher level of abstraction.
- Many manipulations apply lemmas from the prelude or its extensions. Strategies enable proving without explicit knowledge of these lemmas.
- The brittleness of proofs (breakage caused by changes in definitions or lemmas) is reduced by avoiding the inclusion of expressions from the current sequent in stored proof steps.
- Proving becomes more approachable for those with mathematical sophistication but little experience using mechanical provers.

We envision some features as being more useful during later stages of proof development, especially when finalizing a proof to make the permanent version more robust. During the early stages, it is easier to work directly with actual expressions. Once the outline of a proof is firm, extended expression features can be introduced to abstract away excessive detail.

3 Domain-Specific Strategies

Systematic strategy development for various domains could improve user productivity considerably. This section proposes a general scheme for structuring and implementing strategies in PVS and briefly sketches a particular set of strategies for manipulating arithmetic expressions.

3.1 Design Considerations

Input to the PVS prover is via Lisp s-expressions. Internally the prover uses CLOS (Common Lisp Object System) classes to represent expressions and other data. PVS provides macros for creating user-defined proof rules, which may include fragments of Lisp code to compute new values for invoking other rules.

We suggest the following guidelines for developing a strategy package.

1. *Introduce domain-relevant arguments.* For arithmetic strategies, a user typically needs to specify values such as the *side* of a relation (`L`, `R`), the *sign* of a term (`+`, `-`), and *term numbers*. Variations on the conventions of existing prover input handle these cases nicely.
2. *Augment term access functions.* Besides the access functions provided by the prover, additional ones may be needed to extract relevant values, e.g., the ith term of an additive expression. A modest set of access functions suffices for working with common language elements, such as arithmetic terms.
3. *Use text-based expression construction.* A proper implementation style would be to use object constructors to create new expression values. This requires knowledge of a large interface. Instead, it is adequate for most uses to exploit the objects' print methods and construct the desired expressions in textual form, which can then be supplied as arguments to other proof rules.
4. *Use Lisp-based symbolic construction.* To build final proof rules for invocation, the standard Lisp techniques for s-expression construction, such as backquote expressions, work well.
5. *Incorporate prelude extensions as needed.* When prelude lemmas are inadequate to support the desired deductions, a few judiciously crafted lemmas, custom designed for specific strategies, can be added invisibly.

Applications of items 1–4 are demonstrated in the simple example of Fig. 2. Most strategies are rather more complicated than this example, often requiring the services of auxiliary Lisp functions and intermediate helper strategies.

An example of a prelude extension lemma of the sort described in guideline (5) is the following:

```
div_mult_pos_neg_lt1: LEMMA
    z/n0y < x IFF IF n0y > 0 THEN z < x * n0y ELSE x * n0y < z ENDIF
```

```
(DEFSTEP has-sign (term &optional (sign +) (try-just nil))
  (LET ((term-expr (ee-obj-or-string (car (eval-ext-expr term))))
        (relation (case sign
                    ((+) '>) ((-) '<) ((0) '=)
                    ((0+) '>=) ((0-) '<=) ((+-) '/=) (t '>)))
        (case-step `(CASE ,(format nil "~A ~A 0" term-expr relation)))
        (step-list
          (list '(SKIP) (try-justification 'has-sign try-just))))
    (SPREAD case-step step-list))
  "Try claiming that a TERM has the designated SIGN (relationship to 0).
Symbols for SIGN are (+ - 0 0+ 0- +-), which have meanings positive,
negative, zero, nonnegative, nonpositive, and nonzero.  Proof of the
justification step can be tried or deferred.  Use TRY-JUST to supply
a step for the justification proof or T for the default rule (GRIND)."
  "~%Claiming the selected term has the designated sign")
```

Fig. 2. Sample strategy built using PVS `defstep` macro

This lemma simply combines two existing lemmas in prelude theory `real_props` into a conditional form to allow rewriting for any nonzero divisor. In ordinary settings, rewriting to such a conditional expression is likely to be undesirable. In this case, however, the lemma accommodates rewriting plus follow-up steps such as case splitting.

Following the design guidelines above will lead to strategies that are sound by construction. Prover objects are examined but not modified. Proof steps are obtained by expanding the strategies into rule applications for execution by the prover. New PVS expressions are submitted through the parser and typechecked. There are no mechanisms to enforce these good intentions, however. Coding errors could have unintended consequences, but with proper care there should be no side effects on the proof state.

3.2 Algebraic Manipulation Strategies

Users often want to manipulate expressions in the familiar style of conventional algebra, as one would do on paper. We now present a brief sampling of an arithmetic package to support this goal. Selected strategies are discussed that illustrate typical design choices. Appendix A lists the primary strategies in this family. Full details are available in a technical report [8] and user's manual [9].

– `move-terms` *fnum side* `&optional` (*term-nums* `*`)

  With `move-terms` a user can move a set of additive terms numbered *term-nums* in relational formula *fnum* from *side* (L or R) to the other side, adding or subtracting individual terms from both sides as needed. *term-nums* can be specified in a manner similar to the way formula numbers are presented to the prover. Either a list or a single number may be provided, as well as the symbol "`*`" to denote all terms on the chosen side. Example: invoking `(move-terms 3 L (2 4))` moves terms 2 and 4 from the left to the right side of formula 3.

– `cross-mult` `&optional` (*fnums* `*`)

  To eliminate divisions, `cross-mult` may be used to explicitly perform "cross multiplication" on one or more relational formulas. For example, $a/b < c/d$ will be transformed to $ad < cb$. The strategy determines which lemmas to apply based on the relational operator and whether negative divisors are involved. Cross multiplication is applied recursively until all outermost division operators are gone.

– `cancel` `&optional` (*fnums* `*`) (*sign* `nil`)

  When the top-level operator on both sides of a relation in *fnums* is the same operator drawn from the set $\{+, -, *, /\}$, `cancel` tries to eliminate common terms using a small set of rewrite rules and possible case splitting. Cancellation applies when *fnum* has the form $x \circ y \; R \; x \circ z$ or $y \circ x \; R \; z \circ x$. In the default case, when *sign* is `NIL`, x is assumed to be (non)positive or (non)negative as needed for the appropriate

rewrite rules to apply. Otherwise, an explicit *sign* can be supplied to force a case split so the rules will apply. If *sign* is `+` or `-`, x is claimed to be strictly positive or negative. If *sign* is `0+` or `0-`, x is claimed to be nonnegative or nonpositive. If *sign* is `*`, x is assumed to be an arbitrary real and a three-way case split is used. Example: `(cancel 3 0+)` tries to cancel from both sides of formula 3 after first splitting on the assumption that the common term is nonnegative.

– `factor` *fnums* `&optional` (*side* `*`) (*term-nums* `*`) (*id?* `nil`)
`factor!` *expr-loc* `&optional` (*term-nums* `*`) (*id?* `nil`)

If the expression on *side* of each formula in *fnums* has multiple additive terms, `factor` may be used to extract common multiplicative factors and rearrange the expression. The additive terms indicated by *term-nums* are regarded as bags of factors to be intersected for common factors. Terms not found in *term-nums* are excluded from this process. In the `!`-variant, the *expr-loc* argument supplies a location reference to identify the target expression so that it may be factored in place. As an example, suppose formula 4 has the form

```
f(x) = 2 * a * b + c * d - 2 * b
```

and the command "`(factor 4 R (1 3))`" is issued. Then the strategy will rearrange formula 4 to:

```
f(x) = 2 * b * (a - 1) + c * d
```

We provide several strategies for manipulating products or generating new products. This supports an overall approach of first converting divisions into multiplications where necessary, then using a broad array of tools for reasoning about multiplication. Three examples follow.

– `permute-mult` *fnums* `&optional` (*side* `R`) (*term-nums* `2`) (*end* `L`)

For *end* = `L`, the action of `permute-mult` is as follows. Let the expression on *side* of a formula in *fnums* be a product of terms, $P = t_1 * \ldots * t_n$. Identify a list of indices I (*term-nums*) drawn from $\{1, \ldots, n\}$. Construct the product $t_{i_1} * \ldots * t_{i_l}$ where $i_k \in I$. Construct the product $t_{j_1} * \ldots * t_{j_m}$ where $j_k \in \{1, \ldots, n\} - I$. Then rewrite the original product P to the new product $t_{i_1} * \ldots * t_{i_l} * t_{j_1} * \ldots * t_{j_m}$. Thus the new product is a permutation of the original set of factors with the selected terms brought to the left. For *end* = `R`, the selected terms are placed on the right. Example: `(permute-mult 3 L (4 2))` rearranges the product on the left side of formula 3 to be `t4 * t2 * t1 * t3`, with the default association rules making it internally represented as `((t4 * t2) * t1) * t3`.

– `mult-eq` *rel-fnum eq-fnum* `&optional` (*sign* `+`)

Given a relational formula $a\ R\ b$ and an antecedent equality $x = y$, `mult-eq` forms a new antecedent or consequent relating their products, $a * x\ R\ b * y$. If R is an inequality, the *sign* argument can be set to one of the symbols in {`+, -, 0+, 0-`} to indicate the polarity of x and y. Example: `(mult-eq -3 -2 -)` multiplies the sides of formula -3 by the sides of equality -2, which are assumed to be negative.

– `mult-ineq` *fnum1 fnum2* `&optional` (*signs* `(+ +)`)

Given two relational formulas *fnum1* and *fnum2* having the forms $a\ R_1\ b$ and $x\ R_2\ y$, `mult-ineq` forms a new antecedent relating their products, $a * x\ R_3\ b * y$. If R_2 is an inequality having the opposite direction as R_1, `mult-ineq` proceeds as if it had been $y\ R_2'\ x$ instead, where R_2' is the reverse of R_2. The choice of R_3 is inferred automatically based on R_1, R_2, and the declared signs of the terms. R_3 is chosen to be a strict inequality if either R_1 or R_2 is. If either formula appears as a consequent, its relation is negated before carrying out the multiplication. Not all combinations of term polarities can produce useful results with `mult-ineq`. Therefore, the terms of each formula are required to have the same sign, designated by the symbols `+` and `-` in argument *signs*. Example: `(mult-ineq -3 -2 (- +))` multiplies the sides of inequality formula -3 by the sides of inequality -2, which are assumed to relate negative and positive values, respectively.

Figure 3 illustrates these strategies by displaying several proof steps for lemma (1) (see Fig. 1).

```
sin_terms_decr.1 :

[-1]  0 < a!1
[-2]  a!1 <= PI / 2
  |-------
{1}   1 > 2 *
          ((1 / (4 * (n!1 * n!1)
                 + 2 * n!1))
            * --1 * a!1 * a!1)

Rule? (PERMUTE-MULT 1 R 2 R)

Permuting factors in selected
expressions, this simplifies to:
sin_terms_decr.1 :

[-1]  0 < a!1
[-2]  a!1 <= PI / 2
  |-------
{1}   1 > 2 * --1 * a!1 * a!1 *
           (1 / (4 * (n!1 * n!1)
                  + 2 * n!1))

Rule? (CROSS-MULT 1)

Multiplying both sides of selected
 formulas by LHS/RHS divisor(s),
this simplifies to:
sin_terms_decr.1 :

[-1]  0 < a!1
[-2]  a!1 <= PI / 2
  |-------
{1}   1 * (4 * (n!1 * n!1) + 2 * n!1)
       > 2 * (--1 * a!1 * a!1)

Rule? (MULT-INEQ -2 -2)

Multiplying terms from formulas -2
 and -2 to derive a new inequality,
this simplifies to:
sin_terms_decr.1 :

{-1}  a!1 * a!1 <= (PI / 2) * (PI / 2)
[-2]  0 < a!1
[-3]  a!1 <= PI / 2
  |-------
[1]   1 * (4 * (n!1 * n!1) + 2 * n!1)
       > 2 * (--1 * a!1 * a!1)
```

Fig. 3. Proof trace fragment for selected steps from Fig. 1

4 Extended Expression Language

Many prover rules accept PVS expressions as arguments, which take the form of literal strings such as "`2 * PI * a!1`". Strategies in our package may be supplied extended expressions as well as the familiar text string form. This works equally well at the command line and within strategy definitions.

The main extensions provided are location references and textual pattern matching. Location references allow a user to indicate a precise subexpression within a formula by giving a path of indices to follow when descending through the formula's expression tree. Pattern matching allows strings to be found and extracted using a specialized pattern language that is based on, but much less elaborate than, regular expressions.

4.1 Location References

In the location reference form `(! <ext-expr> i1 ... in)`, the starting point `<ext-expr>` must describe the location of a valid PVS expression within the current sequent. Usually this is a simple formula number or one of the formula-list symbols $\{+, -, *\}$. The index values $\{i_j\}$ are used to descend the parse tree to arrive at a subexpression, which becomes the final value of the overall reference. Actually, the final value is a list of expressions, which allows for wild-card indices to traverse multiple paths through the tree. Moreover, the index values may include various other forms and indicators used to control path generation.

Location references may be used as arguments for certain strategies where a mere text string is inadequate. For example, the `factor!` strategy can factor an expression in place using this feature even if the target terms appear in the argument to a function. Thus, location references are reminiscent of array or structure references in procedural programming languages.

An example of a simple location reference is `(! -3 2)`, which evaluates to the right-hand side (argument 2) of formula -3. If this formula is "`x!1 = cos(a!1)`", then the string form of the location reference is "`cos(a!1)`". Adding index values reaches deeper into the formula, e.g., `(! -3 2 1)` evaluates to "`a!1`". Breadth can be achieved as well as depth; `(! -3 *)` evaluates to a list containing "`x!1`" and "`cos(a!1)`".

Index values and directives $\{i_j\}$ may assume one of the following forms:

- An integer i in the range $1, \ldots, k$, where k is the arity of the function at the current point in the expression tree. Paths follow the i^{th} branch or argument, returning the argument as value if i is the last index. The symbols `L` and `R` are synonyms for 1 and 2.
- The index value 0, which returns the function symbol of the current expression. If the function is itself an expression, as $f(x)$ in $f(x)(y)$, indices after the 0 will retrieve components of the expression.
- The *wild-card* symbol `*`, which indicates that this path should be replicated for each argument expression, returning values from all n paths.
- A list `(j1 ... jm)` of integers indicating which argument paths should be included for replication, i.e., a subset of the `*` case.
- One of the *deep wild-card* symbols {`-*`, `*-`, `**`}, which indicates that this path should be replicated as many times as needed to visit all nodes in the current subtree. The values returned are the leaf objects (terminal nodes) for `-*`, the nonterminal nodes for `*-`, and all nodes (subexpressions) for `**`.
- A text string serving as a *guard* to select desired paths from multiple candidates. If the current function symbol matches the string, path elaboration continues. Otherwise, the path is terminated, returning an empty list.
- A list `(s1 ... sk)` of strings that serves as a guard by matching each pattern s_i in the manner of Section 4.2.
- A form `(-> g1 ... gk)` that serves as a *go-to* operator to specify a systematic search down and across the subtree until the first path is found having intermediate points satisfying all the guards $\{g_i\}$ in sequence. The form `(->* g1 ... gk)` returns all eligible paths.

Table 1 illustrates the formulation of location references using this notation.

Note that indexing works for both infix and prefix function applications. For arithmetic expressions, special indexing rules result in some "flattening" of the parse tree during traversal. These conventions are more convenient for arithmetic terms and correspond more closely to our usual algebraic intuition for numbering terms. In particular, additive (multiplicative) terms are counted left to right irrespective of the associative groupings that may be in effect. They are treated as if they were all arguments of a single addition/subtraction (multiplication) operator of arbitrary arity.

In practice, not all of the location reference features are likely to be equally useful. We provide a variety of traversal and search mechanisms to ensure some measure of thoroughness. Some users may choose to limit themselves to simple numeric indexing.

Table 1. Examples of location reference expressions applied to the formulas below

Loc. reference	Expr. strings	Loc. reference	Expr. strings
`(! -2)`	`r!1 = 2 * x!1 + 1`	`(! -1 L * *)`	`x!1, r!1, y!1, r!1`
`(! -2 R)`	`2 * x!1 + 1`	`(! 1 R 1 **)`	`sq(x!1 / 4),`
`(! -2 R 1)`	`2 * x!1`		`x!1 / 4, x!1, 4`
`(! -1 L 2 1)`	`y!1`	`(! - "=")`	`r!1 = 2 * x!1 + 1`
`(! 1 R 1)`	`sq(x!1 / 4)`	`(! -2 * "+")`	`2 * x!1 + 1`
`(! -2 *)`	`r!1, 2 * x!1 + 1`	`(! 1 (-> "sq"))`	`sq(x!1 / 4)`
`(! -1 L 2 *)`	`y!1, r!1`	`(! 1 (-> "sq") 1)`	`x!1 / 4`
`(! -1 L * 1)`	`x!1, y!1`	`(! -1 (->* "*"))`	`x!1 * r!1, y!1 * r!1`

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

4.2 Pattern Matching

Each pattern p_j in `(? <ext-expr> p1 ... pn)` is expressed as a text string using a specialized pattern language. Unlike location references, pattern matches usually produce only a text string and lack a corresponding CLOS object for a PVS expression. The patterns $p_1, \ldots, p_n$ are applied in order to the textual representation of each member of the base expression list. In each case, matching stops after the first successful match among the $\{p_j\}$ is obtained. All resulting output strings are collected and concatenated into a single list of output strings.

A pattern string may denote either a *simple* or a *rich* pattern. Simple patterns are easier to express and are expected to suffice for many everyday applications. When more precision is required, rich patterns offer more expressive power.

Simple patterns allow matching against literal characters, whitespace fields, and arbitrary substrings. Pattern strings comprise a mixture of literal characters and meta-strings for designating text fields. Meta-strings denote either whitespace or non-whitespace fields. A whitespace field is indicated by a space character in the pattern. A non-whitespace field is a meta-string consisting of the percent (`%`) character followed by a digit character (0–9), which matches zero or more arbitrary characters in the target string.

Both capturing and non-capturing fields are provided. A capturing field causes the matching substring to be returned as an output. The meta-string `%0` denotes a noncapturing field, while those with nonzero digits are capturing fields. If a nonzero digit d is the first occurrence of d in the pattern, a new capturing field is thereby indicated. Otherwise, it is a reference to a previously captured field whose contents must be matched. Table 2 illustrates the formulation of simple patterns using this notation.

Rich patterns follow the same basic approach as simple patterns, but add features for multiple matching types and multiple text-field types. The match types include full and partial string matching as well as top-down and bottom-up expression matching.

Table 2. Examples of simple pattern matching applied to the formulas below

Pattern	Matching string(s)	Captured fields
`(? 1 "%1(r!1)")`	`sqrt(r!1)`	`sqrt`
`(? 1 "sqrt(sq(%1))")`	`sqrt(sq(x!1 / 4))`	`x!1 / 4`
`(? -2 "r!1 = %1")`	`r!1 = 2 * x!1 + 1`	`2 * x!1 + 1`
`(? 1 "%1(%0) <")`	`sqrt(r!1) <`	`sqrt`
`(? -1 "> %1 - %0")`	`> r!1 - 1`	`r!1`
`(? 1 "%1(%0) < %1(%2)")`	All of formula 1	`sqrt, sq(x!1 / 4)`

```
{-1}  x!1 * r!1 + y!1 * r!1 > r!1 - 1
[-2]  r!1 = 2 * x!1 + 1
  |-------
[1]   sqrt(r!1) < sqrt(sq(x!1 / 4))
```

5 Higher-Order Strategies with Substitution

Extended expressions allow us to capture subexpressions from the current sequent. Next we add a parameter substitution technique to formulate prover commands. To complete the suite, we add higher-order strategies that substitute strings and formula numbers into parameterized commands. These features are intended primarily for command line use. In LCF-family provers, ML scripting can achieve similar effects.

5.1 Parameter Substitution

A parameterized command is regarded as a template expression (actually, a Lisp *form*) in which embedded text strings and special symbols serve as substitutable parameters. The outcome of evaluating extended

expressions is used to carry out textual and symbolic substitutions. Each descriptor computed during evaluation contains a text string and, optionally, a formula number and CLOS object. Descriptors are the source of substitution data while the parameterized command is its target.

Within this framework, we allow two classes of substitutable data: literal text strings and Lisp symbols. The top-level s-expression is traversed down to its leaves. Wherever a string or symbol is encountered, a substitution is attempted. The final command thus produced will be invoked as a prover command in the manner defined for the chosen higher-order strategy. (In Lisp programming terms, this process can be imagined as evaluating a backquote expression with specialized implicit unquoting. It also has some similarities to substitution in Unix shell languages as well as the scripting language Tcl.)

Parametric variables for substitution are allowed as follows. Within literal text strings, the substrings `%1`, ..., `%9` serve as implicit text variables. The substring `%1` will be replaced by the string component of the first expression descriptor. The other `%`-variables will be replaced in order by the corresponding strings of the remaining descriptors.

Certain reserved symbols beginning with the `$` character serve as symbolic parameters. Such symbols are not embedded within strings as are the `%`-variables; they appear as stand-alone symbols within the list structure of the parameterized command. The symbols `$1`, `$2`, etc., represent the first, second, etc., expression descriptors from the list of available descriptors.

Variants of these symbols exist to retrieve the text string, formula number, and CLOS object components of a descriptor. These are needed to supply arguments for built-in prover commands, which are not cognizant of extended expressions. The symbols `$1s`, `$1n` and `$1j` serve this purpose. Aggregations may be obtained using the symbol `$*` and it variants. Table 3 summarizes the special symbols usable in substitutions.

Table 3. Special symbols for command substitution

Symbol	Value
`$1, $2, ...`	nth expression descriptor
`$*`	List of all expression descriptors
`$1s, $2s, ...`	nth expression string
`$*s`	List of all expression strings
`$1n, $2n, ...`	Formula number for nth expression
`$+n`	List of formula numbers (no duplicates)
`$*n`	List of all formula numbers (includes duplicates)
`$1j, $2j, ...`	CLOS object for nth expression
`$*j`	List of all CLOS objects

5.2 Invocation Strategies

Next we describe a set of general-purpose, higher-order strategies. They are not specialized for arithmetic. Some offer generic capabilities useful in implementing other strategies for specific purposes. For each of these strategies, multiple expression specifications may be supplied as arguments. In such cases, each specification gives rise to an arbitrary number of descriptors. All descriptor lists are then concatenated to build a single list before substitutions are performed. Table 4 lists the strategies provided; several are discussed below.

`invoke` *command* `&rest` *expr-specs*

This strategy is used to invoke *command* after applying substitutions extracted by evaluating the expression specifications *expr-specs*.

As an example, suppose formula 3 is

```
f(x!1 + y!1) <= f(a!1 * (z!1 + 1))
```

Then the command

Table 4. Summary of higher-order strategies

Syntax	Function
`(invoke command &rest expr-specs)`	Invoke command by instantiating from expressions and patterns
`(for-each command &rest expr-specs)`	Instantiate and invoke separately for each expression
`(for-each-rev command &rest expr-specs)`	Invoke in reverse order
`(show-subst command &rest expr-specs)`	Show but don't invoke the instantiated command
`(claim cond &opt (try-just nil) &rest expr-specs)`	Claims condition on terms
`(name-extract name &rest expr-specs)`	Extract & name expr, then replace

```
(invoke (case "%1 <= %2") (? 3 "f(%1) <= f(%2)"))
```

would apply pattern matching to formula 3 to create bindings `%1` = `"x!1 + y!1"` and `%2` = `"a!1 * (z!1 + 1)"`, which would result in the prover command

```
(case "x!1 + y!1 <= a!1 * (z!1 + 1)")
```

being invoked. An alternative way to achieve the same effect using location referencing is the following:

```
(invoke (case "%1 <= %2") (! 3 * 1))
```

As another example, suppose we wish to hide most of the formulas in the current sequent, retaining only those that mention the `sqrt` function. We search for all formulas containing a reference to `sqrt` using a simple pattern, then collect all the formula numbers and use them to invoke the `hide-all-but` rule:

```
(invoke (hide-all-but ($+n)) (? * "sqrt"))
```

`for-each` *command* `&rest` *expr-specs*

This strategy is used to invoke *command* repeatedly, with a different substitution for each expression generated by *expr-specs*. The effect is equivalent to applying `(invoke command e_i)` n times.

As an example, suppose we wish to expand every function in the consequent formulas having the string "`cos`" as part of its name. The following command carries this out, assuming there is only one instance per formula.

```
(for-each (expand "%1") (! + *- ("cos") 0))
```

`for-each-rev` *command* `&rest` *expr-specs*

This strategy is identical to `for-each` except that the expressions are taken in reverse order.

Imagine we wish to find all antecedent equalities and use them for replacement, hiding each one as we go. This needs to be done in reverse order because formula numbers will change after each replacement.

```
(for-each-rev (replace $1n :hide? t) (! - "="))
```

`claim` *cond* `&optional` (*try-just* `nil`) `&rest` *expr-specs*

The `claim` strategy is basically the same as the primitive rule `case`, except that the condition is derived using the parameterization technique. The condition presented in *cond* is instantiated by the terms found in *expr-specs*. Argument `try-just` allows the user to try proving the justification step (the second case resulting from the case split).

For example, to claim that a numerical expression lies between two others, we could use something like

```
(claim "%1 <= %2 & %2 <= %3" nil "a/b" "x+y" "c/d")
```

to generate a case split on the formula "`a/b <= x+y & x+y <= c/d`".

Invocation strategies are useful as building blocks for more specialized strategies that users might need for particular circumstances. Extended expressions can support an alternative to the more code-intensive strategy-writing style that requires accessing the data structures (CLOS objects) representing PVS expressions. This alternative can make lightweight strategy writing more accessible to users without a deep background in Lisp programming.

6 Related Work

Tactic-based proving was pioneered by Milner and advanced by many others, beginning with the work on Edinburgh LCF [11]. The introduction of ML and its use for accessing subterms was also introduced in LCF. Constable and his students developed the Nuprl system [6], which included heavy reliance on tactic-based proof techniques. Tactic-based proving also has been used extensively in more recent interactive provers such as HOL [10], Isabelle [16] and Coq [12]. Although much of this has been devoted to low-level automation, there also have been higher level tactics developed.

In the case of PVS, strategy development has not been as much a focus as tactic development has been for provers in the LCF family. Partly this is due to greater use of decision procedures in PVS as well as an increasing emphasis on rewrite rules. For example, Shankar [17] sketches an approach to the use of rewrite libraries for arithmetic simplification. While these methods are certainly helpful, we believe they need to be augmented by proof interaction of the sort we advocate.

Several researchers have developed PVS strategy packages for specialized types of proving. Examples include a mechanization of the TRIO temporal logic [1], a proof assistant for the Duration Calculus [18], and the verification of simple properties for state-based requirements models [7]. A notable example is Archer's account of the TAME effort [2], which has a good discussion on developing PVS strategies for timed automata models and using them to promote "human-style" theorem proving.

In the area of arithmetic strategy packages for PVS, a semi-decision procedure for the field of real numbers [13], which had been developed originally for Coq, was recently ported to PVS. This package is called Field; it achieves simplification by eliminating divisions and rearranging multiplicative terms extensively. Field has been designed to use some Manip strategies for working with multiplication. César Muñoz continues to enhance Field and maintains an active line of development.

Our work on Manip emphasizes applied interactive proving, features for extracting terms from the working sequent, and flexible mechanisms for exploiting such terms. Many PVS strategy approaches stress control issues, giving less attention to the equally important data issues. Only by placing nontrivial term-access facilities at the user interface can the full potential of interactive strategies be realized.

In a typical control-oriented approach, a strategy might have several plausible sets of rules to apply in speculative fashion. If a given try fails to produce results, bracktracking is performed and an alternative is attempted. By placing more emphasis on data or proof state, the strategy can determine which alternative to select based on attributes of the current state. Allowing users to indicate relevant terms from the sequent sharpens the focus even further during interactive proving.

Currently under study are term access features that allow selection by mouse gestures. "Proof by pointing" techniques [3] are examples of applicable methods that can improve usability in this area. Once selected, a term can be matched with an extended expression for locating it. This can be done without burdening the user to derive the extended expression.

7 Conclusion

The Manip arithmetic package has been used experimentally at NASA Langley and made available to the PVS user community. Along with Field [13], it is now being used to prove new lemmas as they are introduced in Langley's PVS libraries [14]. Proofs for the real analysis and vectors libraries, in particular, have made regular use of Manip strategies. As of May 2003, a total of 325 Manip strategy instances were counted in the

proofs distributed as part of the Langley libraries. Further evaluation is needed to gauge effectiveness and suggest new strategies.

Tactic-based theorem proving still holds substantial promise for automating domain-specific reasoning. In the case of PVS, much effort has gone into developing decision procedures and rewrite rule capabilities. While these are undoubtedly valuable, there is still ample room for other advances, particularly those that can leverage the accumulated knowledge of experienced users of deduction systems. Such users are well poised to introduce the wide variety of deductive middleware needed by the formal methods and computational logic communities. Our tools and techniques aim to further this goal.

Future activities will focus on refining the techniques and introducing new strategy packages for additional domains. One domain of interest is reasoning about sets, especially finite sets. We expect that ideas from the arithmetic strategies can be readily adapted.

Acknowledgments

The need for this package and many initial ideas on its operation were inspired by Ricky Butler of NASA Langley. Additional ideas and useful suggestions have come from César Muñoz of NIA, and John Rushby and Sam Owre of SRI.

References

1. A. Alborghetti, A. Gargantini, and A. Morzenti. Providing automated support to deductive analysis of time critical systems. In *Proc. of the 6th European Software Engineering Conf. (ESEC/FSE'97)*, volume 1301 of *LNCS*, pages 221–226, 1997.
2. Myla Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000.
3. Y. Bertot, G. Kahn, and L. Théry. Proof by pointing. In *Proc. of Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *LNCS*, 1994.
4. R.W. Butler, V. Carreño, G. Dowek, and C. Muñoz. Formal verification of conflict detection algorithms. In *Proceedings of the 11th Working Conference on Correct Hardware Design and Verification Methods CHARME 2001*, volume 2144 of *LNCS*, pages 403–417, Livingston, Scotland, UK, 2001.
5. Víctor Carreño and César Muñoz. Aircraft trajectory modeling and alerting algorithm verification. In *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, pages 90–105, 2000.
6. R. L. Constable and et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
7. Ben L. Di Vito. High-automation proofs for properties of requirements models. *Software Tools for Technology Transfer*, 3(1):20–31, September 2000.
8. Ben L. Di Vito. A PVS prover strategy package for common manipulations. NASA Technical Memorandum NASA/TM-2002-211647, April 2002.
9. Ben L. Di Vito. *Manip User's Manual, Version 1.1*, February 2003. Complete package available through NASA Langley PVS library page [14].
10. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic*. Cambridge University Press, 1993.
11. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Springer LNCS 78, 1979.
12. INRIA. *The Coq Proof Assistant Reference Manual, Version 7.1*, October 2001.
13. C. Muñoz and M. Mayero. Real automation in the field. NASA/CR-2001-211271 Interim ICASE Report No. 39, December 2001.
14. NASA Langley PVS library collection. Theories and proofs available at `http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html`.
15. Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
16. L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer LNCS 828, 1994.
17. N. Shankar. Arithmetic simplification in PVS. Final Report for SRI Project 6464, Task 15; NASA Langley contract number NAS1-20334., December 2000.
18. J. Skakkebaek and N. Shankar. Towards a duration calculus proof assistant in PVS. In *Third Intern. School and Symp. on Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *LNCS*, 1994.

A Algebraic Manipulation Strategies

The following list summarizes the set of manipulation strategies. A few variants have been omitted in the interest of brevity.

Syntax	Function
`(swap lhs operator rhs &opt (infix? T))`	$x \circ y \Longrightarrow y \circ x$
`(group term1 operator term2 term3 &opt (side L) (infix? T))`	L: $x \circ (y \circ z) \Longrightarrow (x \circ y) \circ z$ R: $(x \circ y) \circ z \Longrightarrow x \circ (y \circ z)$
`(swap-group term1 operator term2 term3 &opt (side L) (infix? T))`	L: $x \circ (y \circ z) \Longrightarrow y \circ (x \circ z)$ R: $(x \circ y) \circ z \Longrightarrow (x \circ z) \circ y$
`(swap-rel &rest fnums)`	Swap sides and reverse relations
`(equate lhs rhs &opt (try-just nil))`	$\ldots lhs \ldots \Longrightarrow \ldots rhs \ldots$
`(has-sign term &opt (sign +) (try-just nil))`	Claims term has sign indicated
`(mult-by fnums term &opt (sign +))`	Multiply both sides by term
`(div-by fnums term &opt (sign +))`	Divide both sides by term
`(split-ineq fnum &opt (replace? nil))`	Split $\leq$ ($\geq$) into $<$ ($>$) and $=$ cases
`(flip-ineq fnums &opt (hide? T))`	Negate and move inequalities
`(move-terms fnum side &opt (term-nums *))`	Move additive terms to other side
`(isolate fnum side term-num)`	Move all but one term
`(isolate-replace fnum side term-num &opt (targets *))`	Isolate then replace with equation
`(cancel &opt (fnums *) (sign nil))`	Cancel terms from both sides
`(cancel-terms &opt (fnums *) (end L) (sign nil) (try-just nil))`	Cancel speculatively & defer proof
`(op-ident fnum &opt (side L) (operation *1))`	Apply operator identity to rewrite expression
`(cross-mult &opt (fnums *))`	Multiply both sides by denom.
`(cross-add &opt (fnums *))`	Add subtrahend to both sides
`(factor fnums &opt (side *) (term-nums *) (id? nil))`	Extract common multiplicative factors from additive terms given
`(transform-both fnum transform &opt (swap nil) (try-just nil))`	Apply transform to both sides of formula
`(permute-mult fnums &opt (side R) (term-nums 2) (end L))`	Rearrange factors in a product
`(name-mult name fnum side &opt (term-nums *))`	Select factors, assign name to their product, then replace
`(recip-mult fnums side)`	$x/d \Longrightarrow x * (1/d)$
`(isolate-mult fnum &opt (side L) (term-num 1) (sign +))`	Select a factor and divide both both sides to isolate factor
`(mult-eq rel-fnum eq-fnum &opt (sign +))`	Multiply sides of relation by sides of equality
`(mult-ineq fnum1 fnum2 &opt (signs (+ +)))`	Multiply sides of inequality by sides of another inequality
`(mult-cases fnum &opt (abs? nil) (mult-op *1))`	Generate case analyses for products
`(mult-extract name fnum &opt (side *) (term-nums *))`	Extract selected terms, name replace them, then simplify